Variables, Types, Type Checking

author nlahn@radford.edu; modified by ibarland@radford.edu.

Today

- Defining a few basic concepts that have to do with variables / types
    - These apply to many different languages
- Explicit vs. Implicit. Conversion
- Static vs. Dynamic
- Inferred types
- Immutable vs. Mutable

What is a variable?

What is a variable?

- A named bucket (bucket holds a value; name is on outside of bucket)
- Your code mentions the name in an expression; the computer gets the value from the corresponding bucket.
- Your code can change the bucket's value with assignment (*e.g.* `n = 2+3`);
  general syntax: *`<Var> = <Expr>;`*
- In rust, every variable has a *type*;
  the type determines the bucket's shape (what *type* of values it can hold)

What is a type?

What is a type?

- A type is the set of possible values a variables can hold.
- Internally, compiler knows the `size_of` each type (the size of the bucket -- how many bytes are needed to put one on the stack or heap).
- Internally: A type also implies that the bits representing it will be interpreted in a certain way.
- Example : `let mut x : i32 = 4;`
  - The value of `x` might change, but we know it's always holding some `i32`.
  - The actual data stored in this variable is 32 bits (here, `00…0100`).
  - Interpret those 32 bits as two's complement.

author nlahn@radford.edu; modified by ibarland@radford.edu.

Declaring Variables

- Name *followed* by type:

-
  ```
  let x : i32 = 3;
  println!( "x holds {} right now.", x);
  ```

-
  ```
  let isHappy : bool = true;
  ```

-
  ```
  let numStudents : u16 = 18;
  ```

- General syntax:
  ```
  let <Id> : <Type> = <Expr>;
  ```

Declaring Variables (cont.)

- Just as in Java, it's allowed to declare a variable w/o initializing it.
  But it must be initialized (later) before its first use.
- Good style: always initialize.
  This is required for this course.
  (Some languages don't even allow declare-without-initialize!)

- If initial-value depends on if, you can use if-as-expression:

  ```
  let x : i32 = if a>b {a} else {b};
  ```

- In general `if` is an expression in rust, not a mere statement.
  So it can be used as part of bigger expressions:
  `let x : i32 = 3 + (if a>b {a} else {b})*17;` !!

What is a variable? (internal considerations)

- Internally: name turns into a memory-location after compiling, perhaps as an offset from the current stack-frame.
- Internally: the type determines the *size* of the bucket (how many bytes the item fills, beyond the starting-memory-location)
- Thus at compile-time, rust knows how many bytes are needed for the stack of every function-call. (Very important, for compiling!)

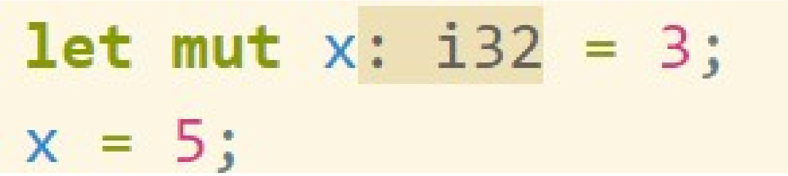
What is a type? (cont.)

- Each type also has a corresponding set of operations
- These may be defined as part of the type definition, or separately.
  - Usually, the operations are functions ("methods"), but can also be operators (which are morally functions, but called infix rather than prefix).

- Ex: A Java class has:
  - Fields: the sub-types comprising the overall class-type.
  - Methods: defines the allowed operations on objects of this class/type.

Immutable vs. Mutable Variables

- By default, *all* Rust variables are immutable
- That means you *can’t assign a new value to them*
- To make a mutable version of a variable, use the **`mut`** keyword

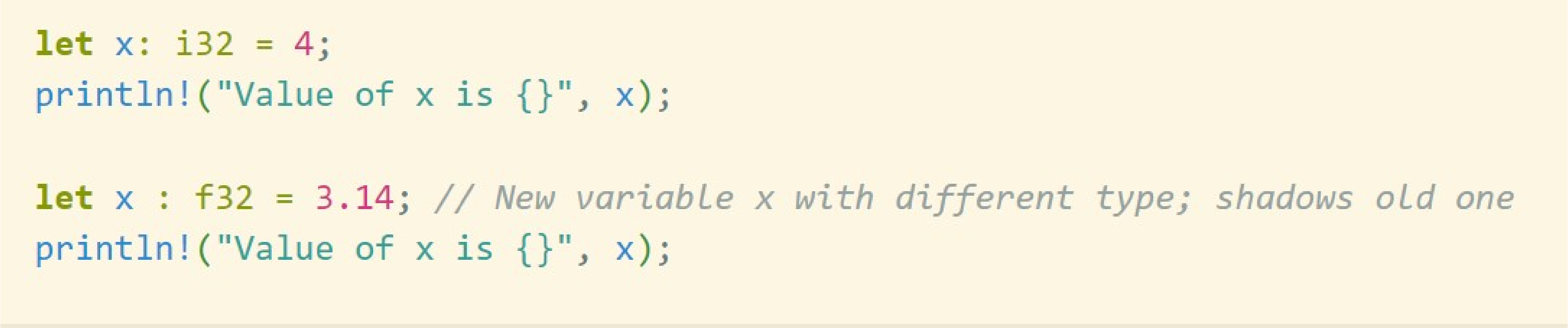


```
let mut x: i32 = 3;
x = 5;
```

- This is the opposite approach from Java, where variables are mutable *unless* you add a keyword: `final`.
- Immutable variables are preferred when possible –
  they make debugging much easier!

Shadowing

- Rust allows “shadowing” of variables:
  two different variables that happen to have the same name.



```
let x: i32 = 4;
println!("Value of x is {}", x);

let x : f32 = 3.14; // New variable x with different type; shadows old one
println!("Value of x is {}", x);
```

- The two variables might even have a different type
- This is *not* somehow changing the type or mutability of a variable:
  - Rather, an *entirely new* variable is made with the *same name*
  - The old variable is unusable: we say it is *shadowed*

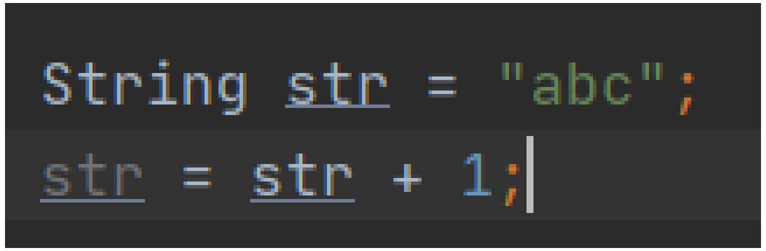
Implicit *vs.* Explicit Conversion

- Conversion: Replace a value of one type with a value of another type

- Explicit conversion : You have to explicitly tell the program to do the conversion
  - Sometimes known as “type casting”
  - It's a function (input: bits-for-int; return: bits-for-double);
    sometimes uses special syntax instead of function-call.
- Implicit conversion : The conversion-function is called automatically, without you having to say anything
  - Sometimes known as type “coercion”

Examples:

- Java uses type coercion for `String` addition (and for `String`s, in general):

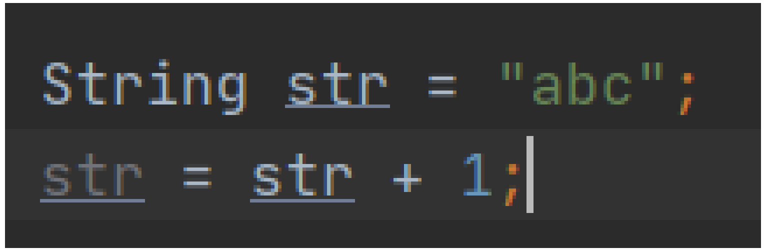


```
String str = "abc";
str = str + 1;
```

- `"Wall is " + Math.sqrt(25) + "ft."`
  What implicit conversion(s) are happening?

Examples:

- Java uses type coercion for `String` addition (and for `String`s, in general):



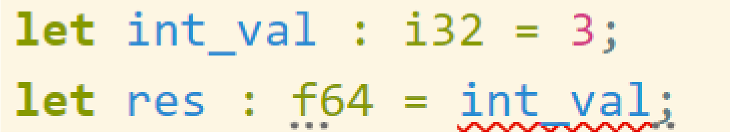
```
String str = "abc";
str = str + 1;
```

- `"Wall is " + Math.sqrt(25) + "ft."`

- `"Wall is " +` **`doubleToString(`**`Math.sqrt(`**`int2double(`**`25`**`))`**`)` `+ "ft."`
- Java privileges `String`s and arrays. Other languages are far more lax: E.g. php: an empty array can be `false` in a boolean context, as can an empty string or `0` (but not "0" even though it `"0"` gets coerced to `0` in numeric contexts).

Examples:

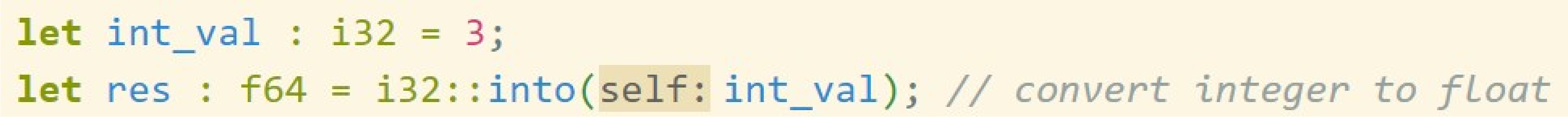
- Rust: Generally requires most conversions to be explicit



```
let int_val : i32 = 3;
let res : f64 = int_val;
```

Error : Can't implicitly convert
from integer to floating point

- To do any operations between different types, must make sure the types match first
  - Can use the From / Into functions



```
let int_val : i32 = 3;
let res : f64 = i32::into(self: int_val); // convert integer to float
```

Strong vs. Weak Typing

Strong v[illegible]ak Typing

- People use “strong” vs. “weak” to mean different things:
  - Sometimes "few coercions" vs. "lots of coercions";
  - Sometimes "statically typed" vs "dynamically typed"
- Avoid using these terms
- Instead of strong/weak, it’s better to talk about implicit vs. explicit conversion

Rust is Statically Typed

- Variables in Rust are statically associated with a type

  That means:
  - The type is known at compile time
  - The type can never change
  - Thus Rust always knows the *size* needed to hold or pass that variable.

- There are a few exceptions involving polymorphism, but they are rare (and complicated)

Comparison: Static Vs. Dynamic typing

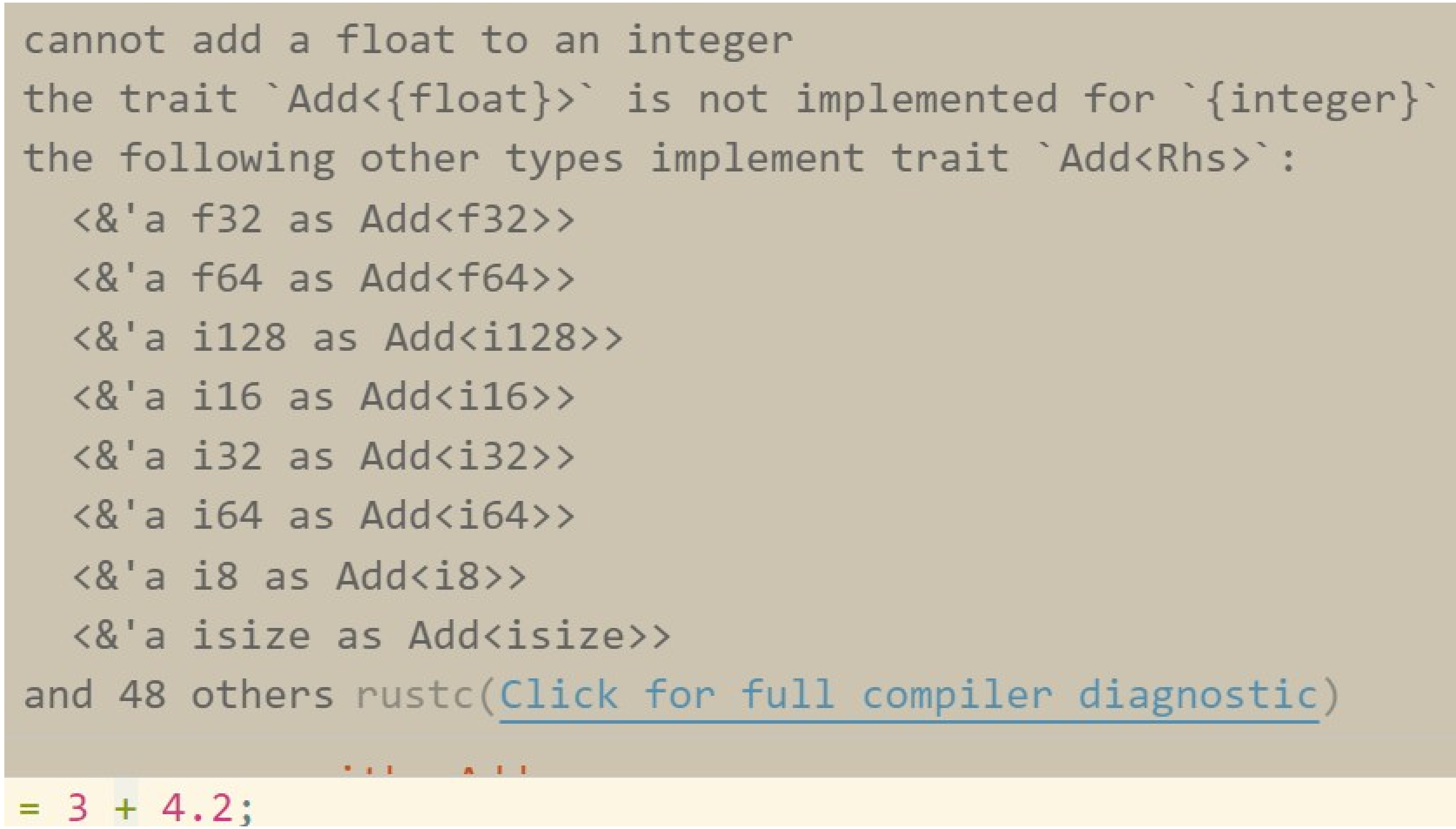
- Generally, statically handling something is *faster* and *safer*
  - Faster : Fewer checks needed at runtime
  - Safer : Error detected when program is compiled; no debugging needed.

    (That is: if it compiles, the compiler has *proven* there are no type-errors!)

- Dynamic handling is sometimes necessary though for flexibility.
  An error might lurk, if your unit-tests weren't thorough.

Rust examples

- Can’t add floats to integer
- Can’t add integers to strings
- Can’t convert between floats and integers, unless you use explicit conversion (like From / Into)



```
cannot add a float to an integer
the trait `Add<{float}>` is not implemented for `{integer}`
the following other types implement trait `Add<Rhs>`:
  <&'a f32 as Add<f32>>
  <&'a f64 as Add<f64>>
  <&'a i128 as Add<i128>>
  <&'a i16 as Add<i16>>
  <&'a i32 as Add<i32>>
  <&'a i64 as Add<i64>>
  <&'a i8 as Add<i8>>
  <&'a isize as Add<isize>>
and 48 others rustc(Click for full compiler diagnostic)

= 3 + 4.2;
```

Static Checking vs. Dynamic Checking

- Static checking done at *compile-time*
  - Examples: Ada, Java, Rust
  - In Java:
    - `String str = "abc";`
    - `str = 5` // Throws an error because `str` has type `String`, and can only hold `String`

- Dynamic checking done at *run-time*
  - Example: Python, Lisp
    - `str = "abc"`
    - `str = 2` // Legal in python
    - `str.split()` // Calling `string` method on `int` not legal; not detectable until run-time
  - *Requires* checks at run-time, hence slower runtime (2x-10x … may not matter)

author nlahn@radford.edu; modified by ibarland@radford.edu.

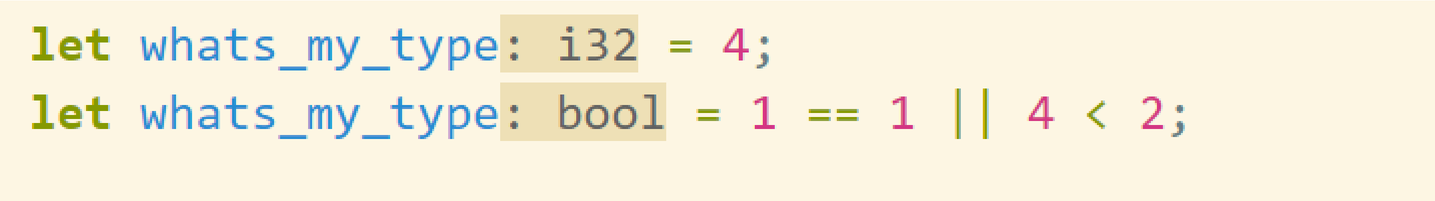
Java polymorphism can be dynamic

- Check out dynamic.java

Inferred vs. Explicit Types

- Rust allows for type inference
- Examples
  - `let whats_my_type = 4;`
    - Automatically inferred as i32
  - `let whats_my_type = 1 == 1 || 4 < 2;`
    - Automatically inferred as `bool`
- If it's ambiguous, Rust compiler won't let you do it

Inferred Types are displayed by VSCode



```
let whats_my_type: i32 = 4;
let whats_my_type: bool = 1 == 1 || 4 < 2;
```

- VSCode’s Rust Analyzer Plugin lists the inferred types in dark yellow
- These are not actually part of the program text
  - They are placed there in the GUI automagically by the editor
- Double clicking the type will actually add it to the text
- I recommend that you explicitly write the type until you understand Rust well